Approaching Oracle Power Objects

An Overview of the New Front-End Tool

By Robert Hoskin

Oracle Power Objects (OPO) is Oracle's answer to Visual Basic, with versions for Microsoft Windows and the Macintosh developed and released simultaneously. Apple features both flavors of OPO in its September *Developer Tools Catalog*. Important Windows standards like OLE2 and OCX are supported, with OpenDoc support and an OS/2 version in the works. Oracle is serious about taking ground back from Microsoft, and leveling the OS and object playing field appears to be an important part of that effort.

Oracle's development tools have always been tightly integrated with their database server, and OPO is no exception. This is a real advantage that saves tons of code and makes it far more likely that your application will scale well. OPO also features native support for Sybase/Microsoft SQL Server, and Blaze, its own scaled down RDBMS that is upwardly compatible with Oracle7.

And although you won't mistake it for Smalltalk, OPO is also object-oriented. Its implementation of visual objects is capable, useful, and it's dead-easy to build your own classes from within the tool. This article will present a whirlwind tour of OPO's underpinnings.

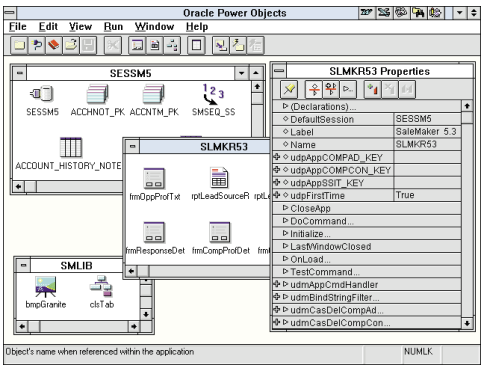

Figure 1: The Oracle Power Objects desktop. An application and its Property sheet are open in the foreground. Session and Library windows are open in the background.

Organization and Terminology

From a developer's perspective, OPO's high-level objects are containers — sessions, applications, and libraries — that exist at design-time as operating system files on your chosen platform. These objects each have their own methods and properties, along with the objects they contain.

Sessions contain and provide access to database objects, such as tables and views. They have their own properties (**DesignConnect** for example) that specify the connection string or path to the database and provide access to the schema (see Figure 1). Sessions also have methods, such as **CommitWork**, **RollbackWork**, and **IsWorkPending**, that return True if a container (a form for example) has uncommitted changes. The database objects that a session "sees" might be seen in Oracle7's USER_CATALOG and USER_INDEXES views. That is, if the necessary tables are in a schema other than the one your session is attached to, they are owned by someone else, and synonyms must be provided for OPO to "see" them.

Applications contain application objects: forms, reports, classes, bitmaps, and OLE objects. Forms, reports, and classes are bindable objects and are usually associated with a database table or view. This close association makes the data-awareness of OPO possible. And applications are more than containers. They have their own properties (e.g. **DefaultSession**) and methods (e.g. **OnLoad** and **CloseApp**), and are a good place to attach user-defined properties and methods that apply across the application.

Like applications, libraries are containers, but their purpose is different. They contain applica-

tion objects that need to be shared across applications. You don't compile or run a library — you reference objects within it. A library's only property is **Name**.

Some application objects are visible objects you apply to forms, reports, and classes to build an application. They are dragged from the Object palette (see Figure 2) and dropped where they belong. These objects can be *static objects* (e.g. static text and lines), *controls* (text fields and scroll bars), or *containers* (a repeater display).

Figure 2: OPO's Object palette.

Most controls and containers are *bindable*, that is, they work closely with an associated underlying table or view. The linkage can be straightforward. For example, a form (container) can be bound to a table, and a field (control) on that form can be bound to a field in the table. Sometimes the linkage is less direct. For instance, all a Current Row Indicator control does is track the current row position in a repeater display. Popup list controls are bound, and reference not only a field in the table a form is bound to, but also the foreign table used for the lookup.

In-memory objects are defined with Oracle Basic at design-time, and created at run-time. There are two "families" of in-memory objects: Recordset objects and "everything else." A *recordset object* is an invisible set of records with a row-and-column structure, as queried from a database session. Recordsets usually hide behind a form or report, but you can also create and manipulate record sets on your own.

The other in-memory objects are *menus*, *toolbars*, and *status lines*.

Sessions and Database Objects

OPO supports several different database Servers: Oracle7, Blaze, Microsoft SQL Server, and Sybase SQL Server. Blaze ships with OPO and is upward-compatible with Oracle7. The chief difference is that Oracle7's PL/SQL and database triggers are not available with Blaze. Since OPO compensates for the minor datatype differences between the two, they are unlikely to make any practical difference in most applications.

OPO's Session window recognizes five database objects: tables, views, synonyms, sequences, and indexes. Simple dialog boxes help to create synonyms, sequences, and indexes, and editors assist with table and view creation.

Sequences, for non-Oracle users, are number servers managed by the RDBMS and provide unique primary key values for a row. The Table Editor window is a grid that allows you to specify the column name, datatype, size, and other attributes of a table column. In the View Editor, you can use drag-and-drop to identify tables, join columns, and create simple views. There are limits to what you can do with drag-and-drop, however. For example, views that use complex SQL statements or functions must be built with another tool.

You can also use drag-and-drop to move objects between schemas. Just open two Session windows, and drag an object from one window to another to add it to the other session. Table objects retain all their rows when you drag them.

Methods and Oracle Basic

You could probably build a simple codeless OPO application by dragging and dropping objects and setting their properties correctly. However, real-world applications are rarely that simple. Most applications need procedural code, whether it sets a few properties, or provides support for complex business rules. OPO uses an extension of ANSI BASIC as its scripting language to modify its built-in methods (see Figure 3), and create user-defined methods.

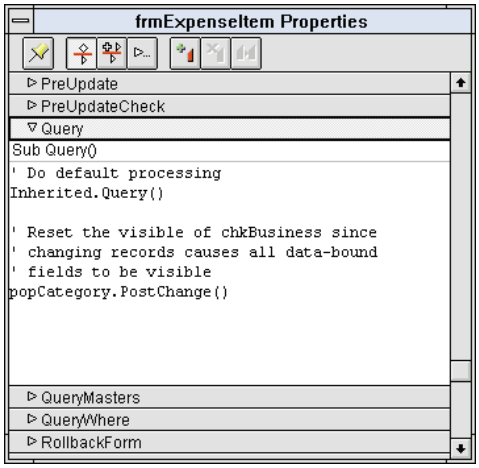

Figure 3: A few lines of Oracle Basic supplementing a **Query** method. The default behavior is inherited, and an additional method is called. This sample is from the PIM sample application.

Some OPO methods are executed by user-originated events, like a mouse-click or key press. Others are only executed when called by custom Oracle Basic code. In many cases, an event causes a series of methods to be executed. For example, when an **InsertRow** method is executed, it sets off not only **PreInsert** and **PostInsert** events on the subject row (just to name two), but also fires **LinkPreInsertCheck**, **LinkPreInsert**, and **LinkPostInsert** events on other objects that are related by key.

These spin-off method calls must be remembered when supplementing one in your own code. The code you add replaces the built-in functionality of the method, which is fine if you want to suppress a method. Usually, you'll want to suppress a method, preserving its behavior and adding an action of your own. You achieve this by adding the statement:

```
Inherited.method_name()
```

to your code. This will execute the default processing for the method.

And don't go looking for diagrams that document OPO event handling and its default method execution sequence. Unfortunately, you won't find them. The topic receives only cursory treatment in the otherwise useful on-line help and *User's Guide*.

You can also define your own properties and methods, and associate them with just about any of OPO's objects, from applications to fields. When defining a method, you must decide if it will be a subroutine (i.e. procedure) or function, the type of value it returns (if it is a function), and the number and type of any arguments it will be passed. This doesn't extend the event model, however, so when you need to execute a method you must call it from within one of the built-in, event-driven methods, such as **Click**.

As a new OPO developer you may chafe at the strict global-or-local scope of Oracle Basic variables. The feeling doesn't last long, however. After a while, it becomes apparent that the object-wide variables you want to implement are really properties of that object, and the solution is to employ a *user-defined property*.

Containers and Recordsets

Containers and recordsets are two central concepts within OPO. Containers hold other objects within them, and may be bound to a database object. Recordsets are in-memory objects that hold rows and columns of data queried from a database. Bound containers have their own recordsets, while freestanding recordsets can be created using Oracle Basic.

Bound containers, such as forms, all have a **RecordSource** property that names the table or view that the container's recordset maps to. This does not mean that a "form" from the user's viewpoint must map to only one table. Containers can contain other containers, so forms can contain embedded forms and other bindable objects. (More on this later.)

One particularly useful side effect of this distinction between a container and its recordset is the ability of bindable containers to share a single recordset. A single form can perform a query and set the context for other forms in an application. Those other forms can share the querying form's recordset by simply setting a property. Change a value in one form, and it's reflected in all. Move to another row in one form, and all the others move as well. Slick.

Bindable controls, such as fields and radio button frames, are used inside bound containers. These controls have a **DataSource** property. If the control is bound, **DataSource** names a column from its container's **RecordSource**, and the control maps to that column in the recordset.

But how do you control the query behavior of a bound container such as a form? If the query criteria doesn't vary, set the form's **DefaultCondition** property with a valid SQL query condition. At run-time, OPO appends it to the recordset's WHERE clause and brings back the records you need. If the query conditions vary, you should use the container's **QueryWhere** method. This overrides the default query condition with one you supply, and executes a query.

There are times when the distinction between containers and recordsets is inconvenient. To use methods that operate down at the recordset level, such as **SetColVal**, you need to first use the container **GetRecordset** method to get a reference (handle) to the recordset. Recordsets are integrated with containers, so if you work with them mainly interactively with drag-and-drop rather than Oracle Basic, it's easy to forget that they're not the same thing.

Standalone recordsets are useful when you perform record-by-record processing behind the scenes. After the few steps it takes to declare a recordset, use the **SetQuery** method. The recordset assumes column definitions that match those in the SELECT statement and sets up the query for execution.

OPO often performs incremental fetches that are useful for presenting information economically. For standalone recordsets, you should call the **FetchAllRows** method. This pulls the whole result set into the recordset in one pass. You can then use an Oracle Basic FOR loop and **SetCurRow** method to step through the recordset.

Forms and Reports

Forms and reports are visible as discrete objects in an Application window. You create a new form in its Designer window by pushing the appropriate toolbar button and then configuring its basic properties (characteristics), such as **Name** and **WindowStyle**, by manipulating its Property sheet.

Most development activities are visual. Drag columns from a session's table onto a new form and OPO will set the form's **RecordSource** property, create the field controls, and set their **DataSource** properties. Drag a scroll bar from the Object palette onto the form, and you have a plain-but-serviceable single-record form that's ready to compile and run. To change a field's default label, click on it, and change its **Label** property. This property is modifiable at run-time, so "soft" prompts are not difficult to implement if your application requires them.

When you build a form by dragging and dropping, the properties that govern database binding are set automatically. For many applications, you may only need to set properties like **Bitmap**, **ColorFill**, **HasBorder**, and **WindowStyle**. These determine the background, border, resizeability, and modality of the form. You may want to set the field-level properties: **DrawStyle**, **HasBorder**, **FontName**, and **FontSize** as well. These control the presence and appearance of a field's border and font. There are 34 field properties that govern appearance, positioning, behavior, updateability, and binding (see Figure 4).

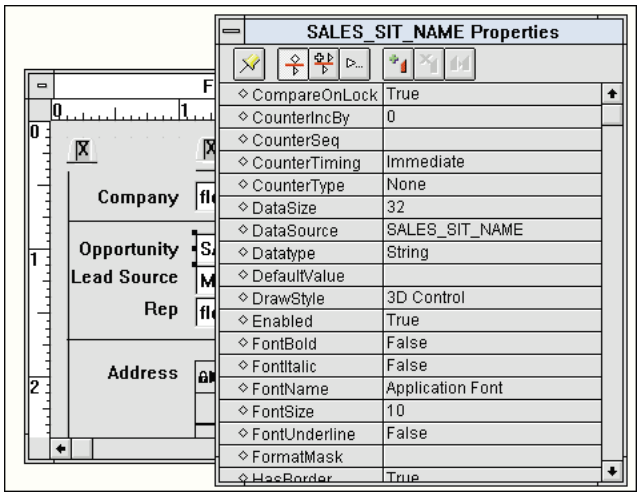

Figure 4: A selection of field-level properties.

OPO reports are similar to forms and function as simple "banded" report writers. A report is a container you design by dropping objects onto it. Page breaks and totals are managed by inserting objects called *report groups*. Properties like **PageOnBreak** and **FirstPgHdr** control pagination and toggle the printing of group headers and footers.

More complex forms use embedded forms or repeaters. Embedded forms, for all practical purposes, are those contained by other forms. When you need one, pick it off the Object palette and drop it on a form. Each has a **RecordSource** property and can contain anything that a form contains.

Forms, embedded forms, and repeaters also share another set of properties: **LinkMasterForm**, **LinkMasterColumn**, and **LinkDetailColumn**. These three properties govern synchronization with recordsets in other forms. **LinkMasterForm** contains the name of the parent form that the child container is to be synchronized with. **LinkMasterColumn** names the key column on that parent form that the container must synchronize with. **LinkDetailColumn** is simply the corresponding key column on

the child container. All you have to do to synchronize two containers is set three properties on the child container.

Putting It Together

So how does all of this fit together? Let's say you want to build a Purchase Order query form named `frmPOquery`, and you have three tables:

- The Purchase_Order_Header table contains a due date, comments, terms, a P.O. number, and a Vendor number.
- The Purchase_Order_Details table contains P.O. line items, quantities, and a P.O. number.
- The Vendor_Master table contains a Vendor number, name, and address.

Then build the form:

- Create a new form by clicking the New Form button on the toolbar. Change the form's **Name** property to `frmPOquery`.
- Drop a selection of fields from the Purchase_Order_Header table onto the empty form and arrange them.
- Then add an embedded form from the Object palette, size it to suit, and name it `frmPOvendor`.
- Drop name and address fields from the Vendor_Master table onto the embedded form.
- Set the `frmPOvendor` form's **LinkMasterForm** property to `frmPOquery`.
- Set the `frmPOvendor` form's **LinkMasterColumn** property to `VENDOR_NUMBER`.
- Set the `frmPOvendor` form's **LinkDetailColumn** property to `VENDOR_NUMBER`.
- Pick a repeater object off the Object palette and drop it on the form.
- Set its **Name** property to `repPOdetail`, and size it to suit.
- Drop Item and Quantity fields from the Purchase_Order_Details table onto the repeater.
- Set the `repPOdetail` repeater's **LinkMasterForm** property to `frmPOquery`.
- Set the `repPOdetail` repeater's **LinkMasterColumn** property to `PO_NUMBER`.
- Set the `repPOdetail` repeater's **LinkDetailColumn** property to `PO_NUMBER`.

When this form is compiled and run, scrolling through the Purchase_Order_Header table will cause the Vendor name and address to synchronize as the vendor key on the P.O. changes. The line items in the repeater will also change, synchronizing with each new purchase order. No code is required — just set a few properties.

Want to calculate the total line items on each P.O.? That's not difficult. Just put a text field from the Object palette on `frmPOquery`, set its **DataSource** property to `=Derived`, and then enter an aggregation function like:

```
=COUNT(repPOdetail.LineItem)
```

Aggregation functions are always used outside the containers they refer to.

Classes

OPO's user-defined classes are bindable container objects. Master classes are created in much the same way as new forms. Master class objects should probably be stored in a library (rather than in an application) to facilitate cross-application reusability, but the choice is the developer's. Master classes can use the same controls as forms and reports. Instances of a class are created by dropping them onto a form or report. These instances inherit any modifications made to the master class, unless overridden locally.

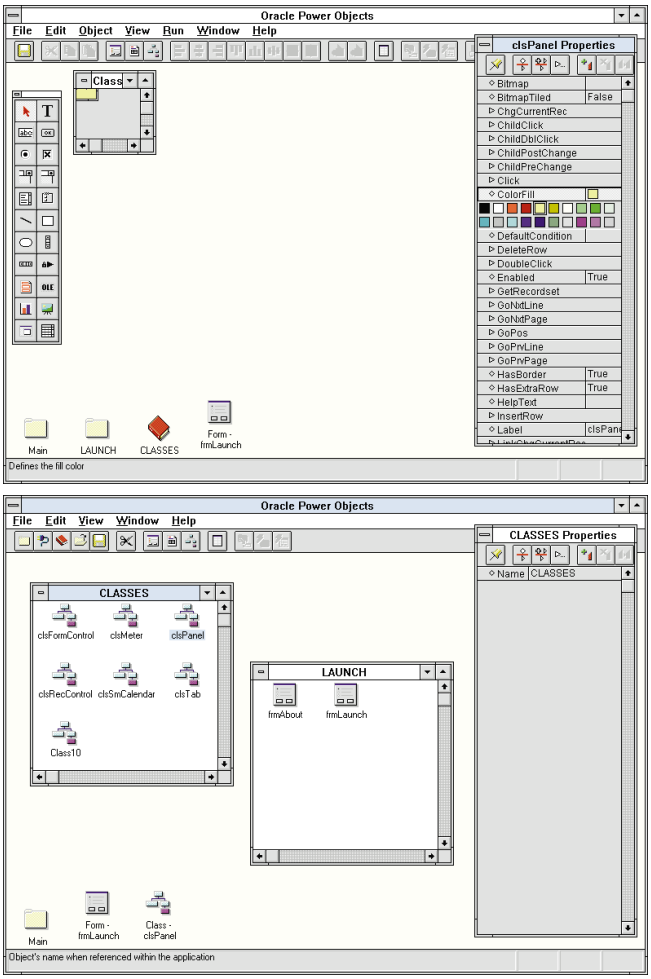

Figure 5 (Top): The `clasPanel` class. **Figure 6 (Bottom):** The `clasPanel` class and `frmLaunch` form on the OPO desktop.

The sample applications that ship with OPO contain many good examples of classes. A simple one from the "Classes" library is named `clsPanel` (see Figure 5). It's used in the sample "Launch" application to draw a bordered panel.

Let's use this class to demonstrate inheritance. To do this, open the `clsPanel` class and `frmLaunch` form (see Figure 6). The name of the panel instance in the `frmLaunch` form is `clsPanelPick`. Note that the panel instance has a **ColorFill** property of light gray.

On the `clsPanel` class' Property sheet, modify the **ColorFill** property to yellow. Notice how the `clsPanelPick` object immediately inherits the color change from its parent object, `clsPanel` (see Figure 7). This is inheritance in action.

Now let's override the **ColorFill** property on the form's Property sheet by selecting gray. Once you've made the color change, note that the small "diamond" next to the **ColorFill** property is now solid. This indicates that the property has been overridden. Whenever a property is overridden in this fashion, it's possible to reinherit the parent class' property value by clicking on the reinherit button at the top far right of the Property sheet. The diamond will revert to its "hollow" aspect and the color will be restored to yellow.

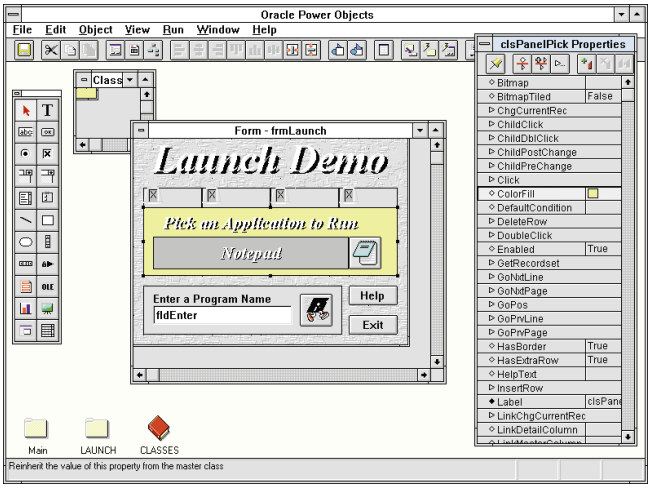

Figure 7: The `clsPanel`'s **ColorFill** property has just been changed to yellow, and the `frmLaunch` form's `clsPanelPick` **ColorFill** property reflects the property change.

This has shown us that instances inherit properties from master classes, and that these properties can be overridden locally. These local overrides can be discarded if necessary by reinheriting them. Instances can also have their own methods and properties in addition to those in the master class. (Remember to discard your changes when you close **clsPanel** and **frmLaunch**.)

Toolbars and Menus

Of course, OPO windows have default menus and toolbars. To navigate a custom application, however, you'll probably want to supplement them.

Menus and toolbars are both in-memory objects, and are created at run-time by OPO. To use them, you define the objects, append entries such as menu lines or tool icons to them, assign the configured object to a form in its **InitializeWindow** method, and then perform certain actions if a menu line or tool is picked.

How is this done? The whole command-execution framework revolves around the notion of a *command code*. Command codes are numeric values (preferably implemented with a symbolic constant) assigned to menu and toolbar entries. When an item is selected, its command code is activated. This change of state is picked up by a method in the window that the menu or toolbar object is assigned to, and method calls are issued to respond to the selection.

The test and execution mechanism is interesting. Two methods, **TestCommand** and **DoCommand** do the work. **TestCommand** is executed many times a second as OPO scans the menu and toolbar for activity. You can place a CASE statement in this method to selectively enable and disable menu and toolbar items based on application conditions, or you can turn everything on with one line of code.

If **TestCommand** receives a command code from an enabled menu or toolbar item that has been selected, it calls **DoCommand** to do the work. The Oracle Basic code in **DoCommand** tests to see which command code has been passed to it and performs the task.

EXEC SQL

If you can develop most applications with almost no code, why do you need EXEC SQL? It's for things that can't be done the easy way! EXEC SQL lets you call stored procedures and execute arbitrary SQL statements such as:

```
INSERT ... AS SELECT ...
```

These probably fit best in the background of an application.

One useful variant on EXEC SQL is the **SqlLookup** function which accepts two arguments: the session and a SELECT statement. It returns a single row, single column result, and is useful for things like picking up the description for a foreign key column.

Cross-Platform Support

As noted earlier, OPO is a cross-platform product. When you use the fonts native to your platform, be careful that the names and sizes match the fonts available on all the platforms you'll be deploying on.

On the Macintosh, OPO does not currently support Apple Events and AppleScript directly, but you can call external resources. On the Windows platform, you can call any DLL by declaring a function or subroutine that points to the DLL. However, should you need to pass a structure to the DLL, you must create a "wrapper" for the DLL. Oracle Basic does not yet support structures.

Compiler

OPO compiles its applications down to p-code that is then executed by a run-time engine on the target platform. At compile-time, you have the choice of producing a run-time application file or a standalone executable (see Figure 8). The former contains only the application and requires an external OPO run-time engine. The latter has OPO's run-time engine bound to the application in a single file.

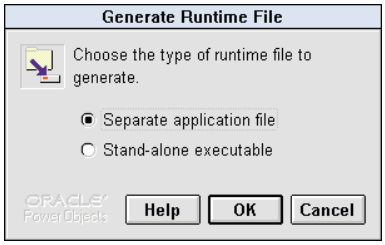

Figure 8: Compiling an application for distribution.

There are tradeoffs here. If a single application is to be distributed to a known target machine, the standalone executable makes sense. If multiple OPO applications are to be distributed to a target machine, application files plus a single copy of OPO's run-time engine is more compact. The application files are cross-platform and can be run under Macintosh or Windows.

Conclusion

Oracle Power Objects is a promising new tool, and certainly welcome to those of us who value built-in data-awareness, as well as cross-platform support. A Visual Basic-like product for the Mac is long overdue. There are some rough edges, but fewer than you might expect from a version 1.0 product. This one is worth a look.

Robert Hoskin is principal of Heartland Software, specializing in training and support of Oracle Power Objects. He can be reached at (905) 858-1415, or 70524.2346@compuserve.com.

Available December 1996

The Must Have Reference Source For The Serious Oracle® Developer

oracle informant

CD ROM

The Entire Text of all Technical Articles Appearing in Oracle® Informant® in 1996

The Oracle® Informant® Works 1996 CD-ROM will include:

- All Technical Articles
- Text and Keyword Search Capability
- Improved Speed and Performance
- All Supporting Code and Sample Files
- 16-Page Color Booklet
- Third-Party Add-In Product Demos
- CompuServe Starter Kit with $15 Usage Credit.

A $130 Value
Available Now for only

$49.95

California residents add 7¼% Sales Tax, plus $5 shipping & handling for US orders.
(International orders add $15 shipping & handling)

Call Now Toll Free
1-800-88-INFORM

1-800-884-6367 Ask for offer # WEB
To order by mail,
send check or Money Order to:
Informant Communications Group, Inc.
ATTN: Works CD offer # WEB
10519 E. Stockton Blvd, Suite 142
Elk Grove, CA 95624-9704
or Fax your order to 916-686-8497

Get Informed!

Subscribe to Oracle Informant, The Independent Monthly Guide to Oracle Development.

Order Now and Get One Issue FREE!

For a limited time you can receive the first issue FREE plus 12 additional issues for only $49.95 That's nearly 25% off the yearly cover price!

Each big issue of *Oracle Informant* is packed with Oracle tips, techniques , news, and more!

- Client/Server Development
- Using Developer/2000™
- Tuning Oracle7
- PL/SQL Techniques
- Advanced Oracle Topics
- Distributed Managment
- Product Reviews
- Book Reviews
- News from the Oracle Community
- Oracle User Group Information

Order Now!

YES!, I want to sharpen my Oracle skills. I've checked the subscription plan I'm interested in below

☐ Magazine-Only Subscription Plan...
13 Issues Including One Bonus Issue at $49.95.

☐ Magazine AND Companion Disk Subscription Plan...
13 Issues and Disks Including One Bonus Issue and Disk at $119.95
The Oracle Informant Companion Disk contains source code, support files, examples, utilities, samples, and more!

☐ Oracle Informant Works 1996 CD-ROM $49.95 (Available December 1996)
US residents add $5 shipping and handling. International customers add $15 shipping and handling.

Name _____

Company _____

Address _____

City _____ State _____ Zip Code _____

Country _____ Phone _____

FAX _____ E-Mail _____

Payment Method...

❑ Check (payable to Informant Communications Group) ❑ Purchase Order-- Provide Number _____

❑ Visa ❑ Mastercard ❑ American Express Card Number _____

Expiration Date _____ Signature _____

WEB

To order, mail or fax the adjoining form or call (916) 686-6610 Fax: (916) 686-8497

International rates

Magazine-only
$54.95/year to Canada
$74.95/year to Mexico
$79.95/year to all other countries

Magazine AND Companion Disk Subscriptions
$124.95/year to Canada
$154.95/year to Mexico
$179.95 to all other countries

California Residents add 7¼% sales tax on disk subscription